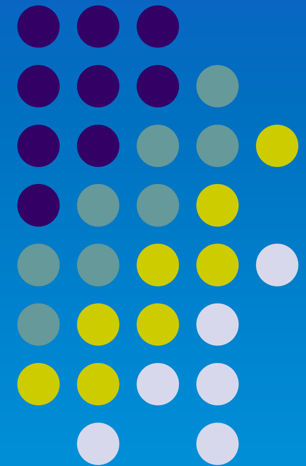
Projektowanie obiektowe Wzorce projektowe



Gang of Four
Behawioralne wzorce projektowe
(Wzorce operacji)

Roadmap

- Strategy
- Template method
- State
- Command

The Sacred Elements of the Faith

the holy origins

the holy behaviors

the holy structures

107 FM Factory Method							139 A Adapter
117 PT Prototype	127 S Singleton				223 CR Chain of Responsibility	163 CP Composite	175 D Decorator
87 AF Abstract Factory	325 TM Template Method	233 CD Command	273 MD Mediator	293 O Observer	243 IN Interpreter	207 PX Proxy	185 FA Façade
97 BU Builder	315 SR Strategy	283 MM Memento	305 ST State	257 IT Iterator	331 V Visitor	195 FL Flyweight	151 BR Bridge

Wzorce behawioralne
Wzorce operacji

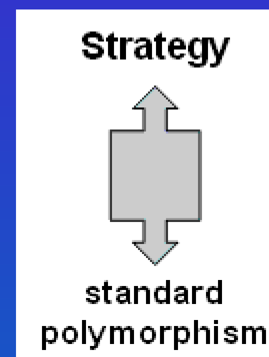
- ***wzorce behawioralne*** umożliwiają organizację, zarządzanie i łączenie zachowań.

- ***wzorce operacji*** (**template method, state, strategy, command, interpreter**) dotyczą głównie sytuacji, gdy w projekcie potrzeba wielu metod zazwyczaj z identyczną sygnaturą.

Pojęcia

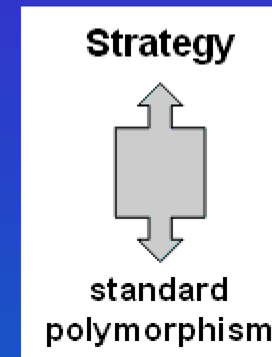
- **algorytm …**
- **operacja …**
- **metoda …**
- **sygnatura …**

Strategy



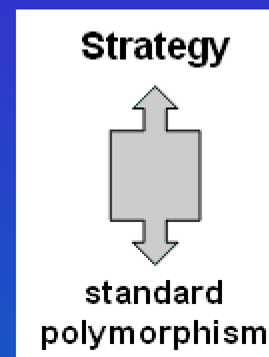
Polega na hermetyzowaniu operacji umożliwiając stworzenie zamiennych implementacji.

Strategy - problem



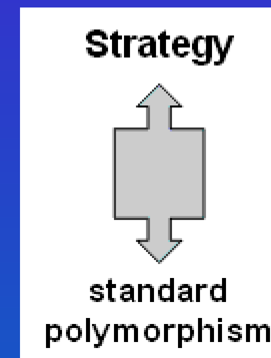
- Złożoność kodu wynikająca z istnienia wielu strategii dotyczących określonego problemu.
- Potrzeba budowy oprogramowania zorientowanego-obiektowo ze zminimalizowaną liczbą zależności.

Strategy - rozwiązanie



- Daj możliwość skonfigurowania wyboru algorytmu
- Struktura osłona/delegacja
  - klient jest osłoną,
  - obiekt algorytm jest delegacją.
- Dodanie poziomu pośredniego dla klienta (np. interfejsu).

Strategy – diagram klas



Client

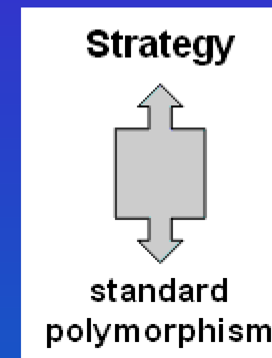
SortAlgorithm
+sort()

BubbleSort
+sort()

HeapSort
+sort()

ShellSort
+sort()

Strategy – przykład



TransportationToAirport

GotoAirport(Time, Cost)

Strategies (Options)

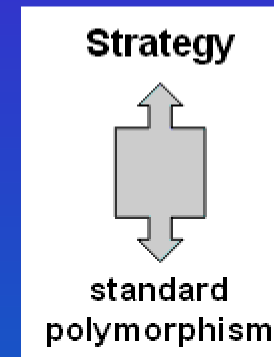
Personal Car

Taxi Cab

Limousine

City Bus

Strategy - konsekwencje

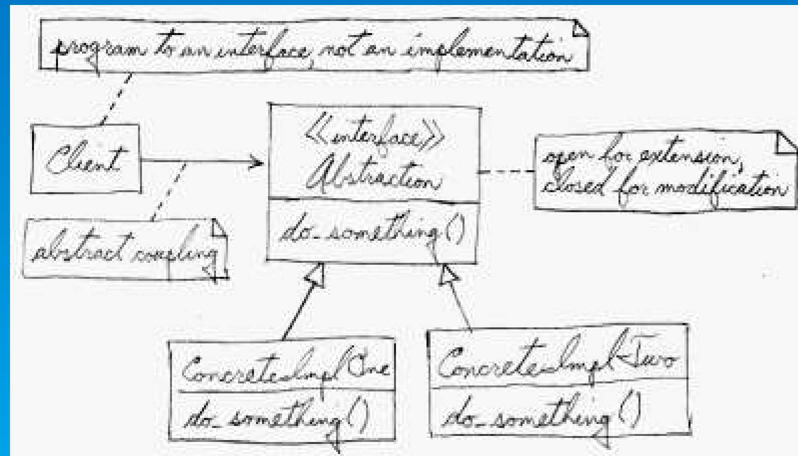


- Zachowanie obiektów klienta może być określone za pomocą obiektów.
- Wzorzec upraszcza klasy klienta przez zwolnienie ich z odpowiedzialności wyboru zachowania lub implementacji alternatywnych zachowań.
- Upraszcza kod dla obiektów klienta poprzez eliminacje instrukcji *if* oraz *switch*. W niektórych przypadkach może zwiększyć szybkość obiektów klienta ponieważ nie potrzebują dokonywać wyboru zachowania.

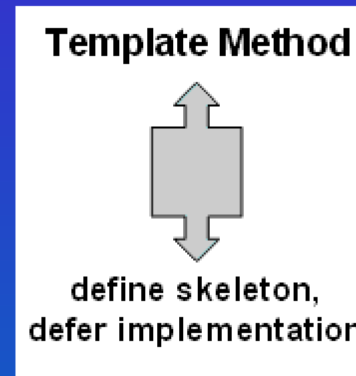
Zasada „otwarcia i zamknięcia"

- **Open-closed principle** [Bertrand Meyer, 1988]**:**

„Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"

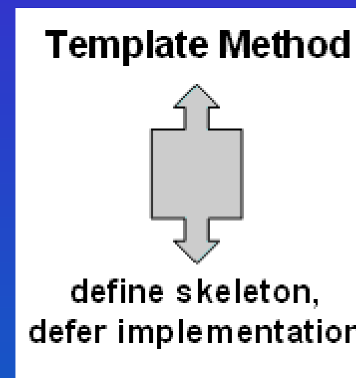


Template Method



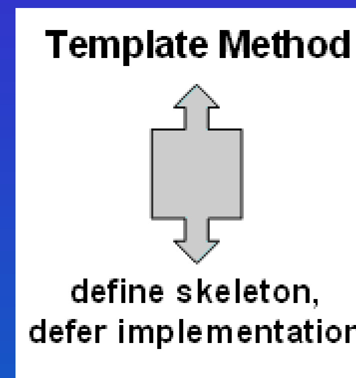
Zaimplementowanie algorytmu (w postaci metody) umożliwiając „opóźnienie” kilku kroków jego wykonania tak, aby klasy podrzędne mogły je ponownie zdefiniować.

Template Method – problem



- Dwa odmienne komponenty mają znaczące podobieństwa, ale nie korzystają z ponownego użycia ani wspólnego interfejsu ani implementacji.
- Jeżeli zmiana wspólnej części staje się konieczna, to niepotrzebnie dublowana jest praca.

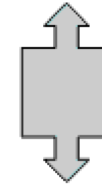
Template Method – rozwiązanie



- Daj możliwość skonfigurowania kroku algorytmu.
- Określany w klasie bazowej krok algorytmu zostawiamy do zaimplementowania w klasach pochodnych.

Template Method – diagram klas

Template Method



define skeleton,
defer implementation

Client

SortAlgorithm2

+sort()
#*compare()*
-processArray()
-returnArray()

processArray();
compare();
returnArray();

SortAscending

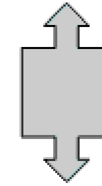
#compare()

SortDescending

#compare()

Template Method – przykład

Template Method



define skeleton,
defer implementation

Basic Floor Plan

- Lay Foundation
- Wire
- Plumb

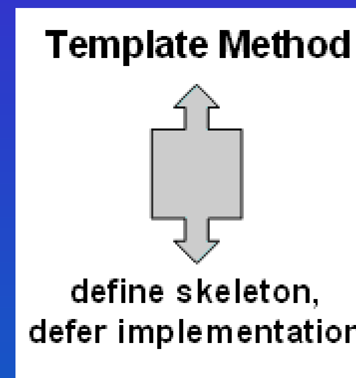
- Add Third Gable
- Add 2 Wings

- Add Wing

- Add Fireplace

Variations added to Template Floor Plan

Template Method – konsekwencje

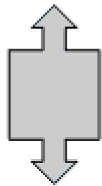


- Programista piszący podklasę abstrakcyjnej klasy szablonu jest zmuszony nadpisać te metody, których implementacja jest konieczna, żeby uzupełnić logikę nadklasy.

- Dobrze skonstruowana klasa szablonu ma strukturę, która dostarcza programiście wskazówki dotyczące podstawowej struktury jej podklas

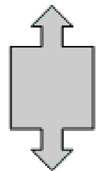
Wzorzec
Strategy czy *Template Method*

Strategy



standard polymorphism

Template Method



define skeleton, defer implementation

Visual Paradigm for UML Standard Edition(Technical University Of Lodz)

Client

Collections

+sort(List list, Comparator cmp)

List

<<Interface>>
Comparator

+compare(Object o1, Object o2)

*

Object

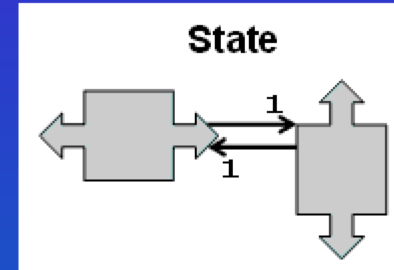
ComparatorImpl1

ComparatorImpl2

Zależności między wzorcami

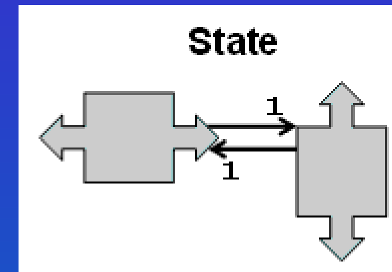
- Dokonuje się zmiany:
  - w części algorytmu poprzez dziedziczenie – ***Template Method***,
  - całości algorytmu poprzez delegacje – ***Strategy***.
- Modyfikowana jest logika:
  - całej klasy – ***Template Method***,
  - indywidualnych obiektów – ***Strategy***.

State



Rozdystrybuowanie operacji na kilka klas w taki sposób, żeby każda klasa reprezentowała różny stan.

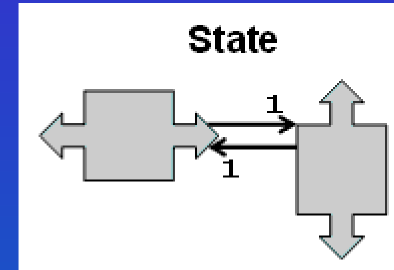
State - problem



- Zachowanie jednolitego obiektu jest zależne od jego stanu.
- Konieczna jest zmiana jego zachowania w czasie wykonania w zależności od bieżącego stanu.

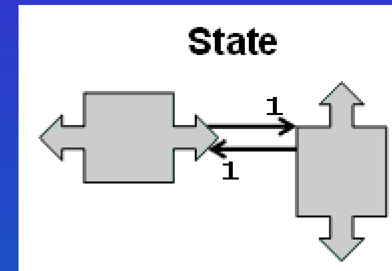
- Aplikacja jest określona przez rozległe i liczne instrukcje warunkowe (*if*, *switch*, etc.), które kierunkują przepływ sterowania w zależności od stanu aplikacji.

State - rozwiązanie



- Struktura osłona/delegacja:
  - osłona przekazuje wskaźnik do siebie („**this**”),
  - delegacja współpracuje z osłoną.

State – diagram klas



Editor

+setMode(newMode : Mode)
+handleKey(theKey : char)

-currentMode

Mode

+handleKey(theKey : char, theEditor : Editor)

currentMode.handleKey(theKey, this);

Insert

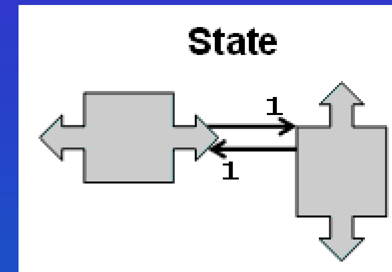
+handleKey()

CommandLine

CommandKey

// collaborate with theEditor object to
//   implement "insert" functionality
// possibly change theEditor object's mode

State – przykład



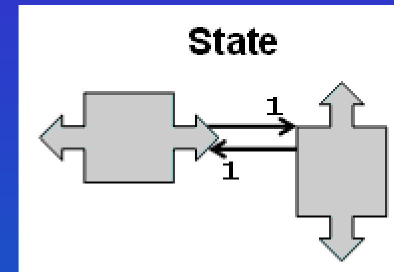
VendingMachineState

VendingDepositState

VendingStockState

ChangeAvailableState

State - konsekwencje



- Kod dla każdego stanu znajduje się w osobnej klasie.
- Można dodawać niezależnie wiele nowych stanów.
- Dla klienta obiektów stanu, przejścia między stanami występują pomiędzy „atomowymi" operacjami.
- Unikamy stosowania instrukcji *switch* lub łańcuchów *if-else* w wielu metodach, przekierowując obsługę do kodu określonego w stanie.
- Nie unikamy jednak instrukcji *switch* lub łańcuchów *if-else* rozdzielających obsługę zdarzenia w ramach bieżącego stanu.

Zależności między wzorcami

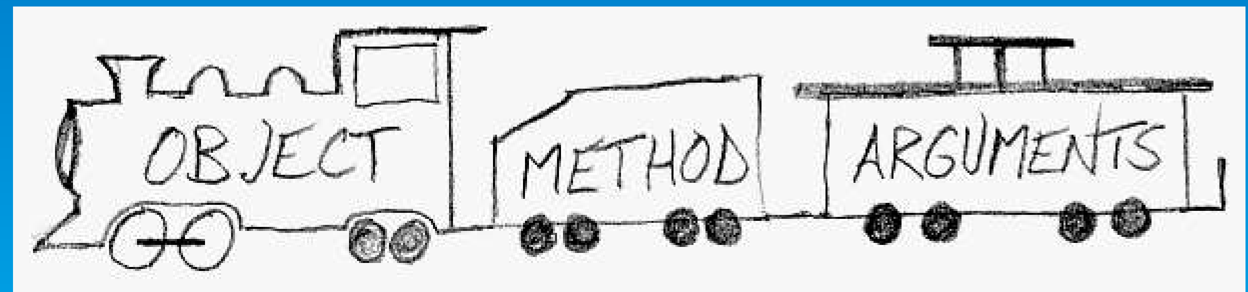
- ***State*** i ***Strategy*** są podobne, ale:
  - ***state*** jest bardziej dynamiczny.

- Struktura wzorców ***State***, ***Strategy***, ***Bridge*** (i trochę ***Adapter***) jest podobna – element uchwyt.

Command

Ma na celu hermetyzowanie wywołania metody w obiekcie.

Umożliwia traktowanie „wywołania metody obiektu” jako pełnoprawnego obiektu (promocja).



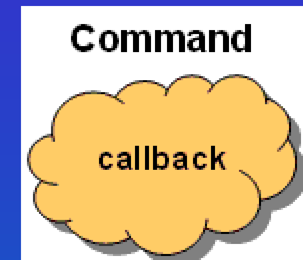
Command - problem

- Potrzeba wydania żądania do obiektów bez żadnej wiedzy na temat:
  - operacji, która jest żądana,
  - lub odnośnie odbiorcy żądania.

Command - rozwiązanie

- Polecenie („**Callback**”) ma być zorientowane-obiektowo, czyli zdefiniuj klasę zawierającą:
    - wskaźnik do obiektu,
    - wskaźnik do funkcji,
    - wszystkie potrzebne argumenty funkcji.
- Zadeklaruj metodę „execute”.
- Zaprojektuj polecenie, aby pełniło rolę „magicznego ciasteczka”, które hermetyzuje wywołanie metody.

Command – diagram klas



Client

CallbackInterface
+execute()

targetObject = receiverObject;
targetMethod = methodPointer;

Receiver
+doThis()
+doThat()

-targetObject

CallbackOne

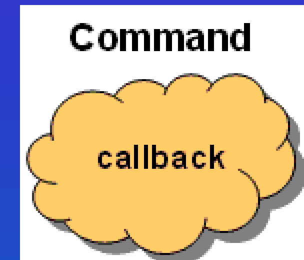
CallbackTwo
-targetMethod
+constructor(receiverObject, methodPointer)
+execute()

```
CallbackInterface token =
        new CallbackTwo( new Receiver(), "doThis" );
... the token object is passed to another object
... and that object calls ...
token.execute();
```

```
// use Java reflection or
// C++ pointer to member function
targetObject.targetMethod();
```

Command – przykład

Command - konsekwencje



- Obiekt, który wywołuje polecenie, nie jest tym samym obiektem, który je wykonuje. Ta separacja umożliwia elastyczne zarządzanie poleceniami (np. kolejkowanie, grupowanie, delegowanie)
- Takie podejście umożliwia „nagrywanie" ciągu poleceń (np. makra) i powtarzanie ich później. Można zastosować do tego wzorzec ***composite***.
- Dodawanie nowych poleceń jest uproszczone, gdyż nie zrywa się żadnych zależności.